# Package Dependency Visualization: Exploration and Rule Generation

Mubarek Mohammed
Department of Computer Science and Electrical Engineering, Syracuse University
Syracuse, USA
mmohamme@syr.edu

James W. Fawcett
Department of Computer Science and Electrical Engineering, Syracuse University
Syracuse, USA
jfawcett@twcny.rr.com

*Abstract*— **Large software systems are difficult to understand and complex to manage. Dependency among software components such as packages is one of the reasons that makes software complex. To partly cope with complexity, we designed and implemented a dependency analysis and manipulation tool to manage package dependency at a higher level. Two major graph layout algorithms, spring layout and Sugiyama layout along with clustering algorithms, are used to visualize layout of dependency graphs. In addition, we propose use of layering with Sugiyama algorithm to get insight about software design. It is also possible to generate dependency rules at the architecture level for software designs that are suited for layering design. Both the visualization and rule generation have flexibility to be used with manual restructuring of package dependency.**

***Keywords- Visualiztion; Dependency; Layering;***

## I. INTRODUCTION

Many organizations report that they face problems managing software complexity using conventional software engineering techniques [1]. Production sized software may contain millions of lines of code [3]. Understanding, changing and maintaining these large software systems is difficult. Documentation may help reduce complexity to some extent. However, as software evolves, changes may not entirely be captured in documents. Up to date characteristics of the software system may be reflected only in the source code. Extracting information about software from source code is, therefore, important.

Software visualization can be used to manage complexity. Software components can be represented as graphical elements such as node and edges. Graph layout as well as analysis of the graph gives insight about the design of a software system. Specially interesting is the use of node-edge graph to represent dependency of packages.

In this work, we show dependency among components, particularly packages, with node-edge graphs to qualitatively examine overall dependency. Using layered graph drawing, edges are discriminated where those going from upper layers to lower layers are considered generally acceptable edges. Whereas those going in the opposite direction are unwanted edges. We simply use two distinct colors: blue and red respectively.

For cases where distinguishing edges does not give useful information, nodes that result in such edges can be clustered into one of the layers. Strong components algorithm is used to achieve this. Further examination of dependency can be done after clustering to see if the resulting dependency graph makes sense. After examining the dependency, developers may restructure the dependency or leave it as it is. The dependency graph can be further clustered by collapsing nodes in each layer into a single node and adding the corresponding edges among the layers. Rules can be generated, as xml file, from the resulting dependency graph. This may be used to restrict further changes to the existing dependency when change is made to the software system.

The main contributions of this paper are:

- Dependency analysis tool that can be used to get insight about a given software project.
- Layering concept that may potentially be used for planning restructuring package dependency.
- Generation of dependency rules that restricts introduction of unwanted changes.

The remaining part of the paper is organized as follows. Section II introduces software visualization and layering concepts. Sections III briefly discusses the layout algorithms used in our visualization tool. Section IV presents the tool design and brief discussion of each component. Section V shows how the tool can be used in real scenario. Section VI reviews related works in software visualization and dependency. Section VII concludes the paper and future extensions of the work.

## II. SOFTWARE STRUCTURE VISUALIZATION AND LAYERING

In this section, we briefly discuss software visualization and layered structure of dependency.

### *A. Software Structrue Visualization*

Software visualization represents software components graphically. Generally, software developers use pictorial representation such as UML diagrams at various stages of software development process. Bringing similar representation at the time of change or maintenance with some interactivity can help understand software better and makes changing or maintenance less complex.

10.18293/DMSVLSS2017-006

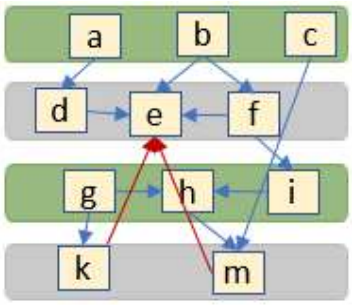


Figure 1. Layering software components.

An important concept in software components representation is dependency. A software component is dependent on another component if the former uses instances or services of the later one.

Once dependency analysis is extracted as node-edge graph, it can be drawn after applying layout algorithms. Nodes represent packages, header and/or cpp file in c++. Edges are added if a package depends on another package. Some layout algorithms such as spring layout show visually appealing diagrams that clarify the existing dependency [9]. Other layout algorithms such as the Sugiyama layout algorithm represent such dependency in a hierarchical way [5]. This helps both in understanding and investigating the meaning of hierarchical representation into layers. The next sub-section briefly introduces layering.

### B. *Layering*

Layering is representing software components in such a way that lower level layers provide services to upper layers [4]. If software is designed in the form of layers, it can be extended by adding upper layers that use services provided by lower layers and contracted by removal of lower layers.

With a strict hierarchy, each layer uses services in the layer immediately below it. With a non-strict hierarchy, a layer does not have to invoke a service at the layer immediately below it, but it can invoke services at more than one layer below.

Layering of software components results in improved maintainability, reusability or testability. General layering concepts can be shown as in figure 1. There are four layers in the hierarchy, layer 1 at the bottom and layer 4 at the top. Most of the arrows go from upper layers to lower layers. From a to d and c to m are two examples. However, there are arrows going from lower level layers to upper layers indicated by red arrows. Arrows from k to e and m to e are two such examples. As will be discussed later, the later kind of arrows are discouraged in a layered design. Layout algorithms used in this work are discussed in the next section.

## III. LAYOUT ALGORITHMS

Layout algorithms arrange nodes in a node-edge graph in such a way that the drawing is visually appealing and readable. Some of the layout algorithms improve symmetry and minimize crossing. Others may arrange the nodes hierarchically.

### A. *Sugiyama Layout Algorithm*

Sugiyama layout algorithm is used for drawing flow charts, UML diagrams and other diagrams that needs to be drawn hierarchically [5]. The algorithm has the following steps:

Step 1 - Cycle removal - removes cycles temporarily as the other steps need an acyclic graph. Algorithms starting from the simple depth first search based feedback edge set to the greedy minimum feedback arc set algorithms can be used to make a graph acyclic [6].

Step 2 - Layering - The nodes are assigned layers based on their dependency hierarchy. Independent nodes take the bottom layer and those depending on lower layers take upper layers. Spanning tree algorithm can be used to assign layers. However, restrictions are applied to width and/or height of the layout. A structure like figure 1 can be obtained using layering.

Step 3 – Cross minimization – this step minimizes the number of crossing between edges of different layers. This is done by repeatedly sorting the layered nodes according to barycentric weights calculated from the index position of nodes in the layers. This makes the drawing visually less cluttered.

Step 4 – Coordinate assignment - x and y coordinates are assigned to each node. The y-coordinate of nodes of each layer will be the same. However, the x-coordinate will be assigned based on the index position of a node in each layer.

### B. *Spring Layout Algorithm*

Force-directed algorithms model a graph layout problem by assigning attractive and repulsive forces between vertices, and finding the optimal layout by minimizing the energy of the system [7], [8]. The model of Fruchterman and Reigold [9], also known as spring-electrical model, has two forces. The repulsive force, exists between any two vertices, and is inversely proportional to the distance between them. Attractive forces, on the other hand exist only between neighboring vertices (vertices that share an arc) and is proportional to the square of the distance. An example is shown in figure 2. For specific situations spring layout gives the most visually appealing graph. Graph interaction discussions that follow, except those that require layering concepts, are applicable for spring layout. Therefore, it will not

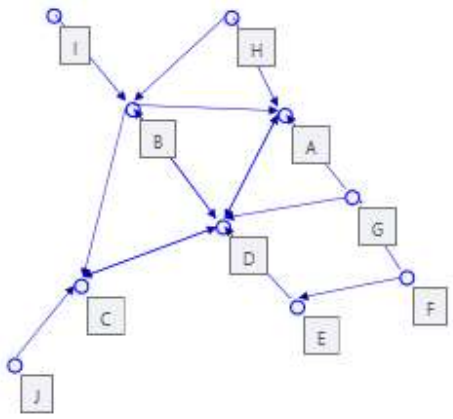


Figure 2. Spring layout example

be discussed further.

Before presenting the visualization tool usage, design of the tool developed will be discussed briefly.

## IV. System Design and Implementation

To study software dependency understanding and quality related issues, a prototype software visualization and analysis tool has been designed and implemented. The system is depicted in figure 3. The major components are discussed as follows.

**Code Analysis** - this component analyzes source code and extracts information such as packages, classes, methods and other software constructs that are not used in this work. This part is implemented based on a light weight C/C++ parser implementation [2]. This is a rule based parser which, with minimal change, may be used to parse other programming languages especially those syntactically similar to C++ such as Java and C#.

**Graph Processing and Layout** - uses information extracted from static analysis as input and results in a node-edge graph. The graph will be further processed to generate the layout. The layout can use spring layout or Sugiyama layout algorithms. In addition, strong components algorithm is used to cluster nodes to modify the layering. Output of this processing is presented as an xml file.

**Visualization** - this component reads the xml representation of the extracted information with coordinates and renders it on a canvas. The user can interact with the visualization tool to see the type of package represented by the node, panning, dimensional zooming and coordinate based zooming.

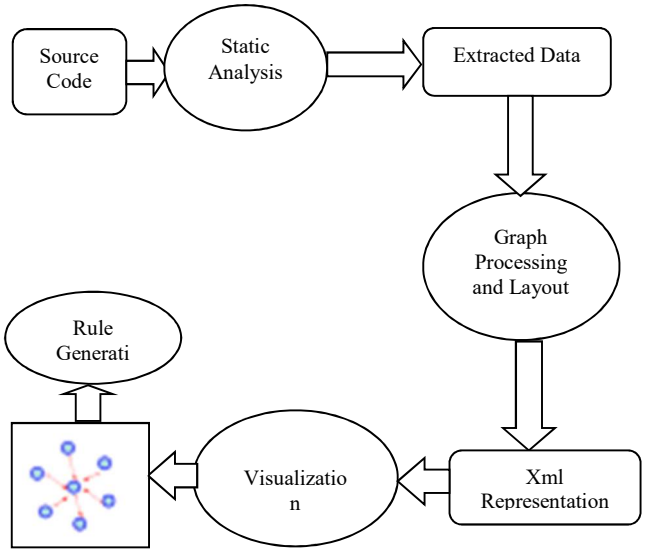


Figure 3. System block diagram.

**Rule Generation** - unwanted dependencies can be generated as xml rules that may be read automatically, at build time, to prevent further deterioration of dependency. These dependencies can be those not shown by arrows going from upper layers to lower layers. It can also be implicit dependencies that go from lower layers to upper layers. The later ones are red edges if they are drawn on the dependency graph.

The static analysis and graph Processing and Layout components are implemented using C++. Whereas the visualization part is implemented using (Windows Presentation Foundation) WPF.

Even though discussion in the sections that follow focuses on package dependency analysis, the visualization system is general enough to be used for other purposes.

## V. Results and Discussions

The dependency analysis tool can be used in different scenarios. Some of its uses are:

- Examining dependency when changing software.
- Getting insight to restructure package dependency.
- Generate rules to prevent addition of unwanted dependency.

We have explored two software systems using the dependency visualization tool:

- Notepad++ - a multipurpose text editor. It has more than 300 packages [10].
- Webkit - taken from chromium web browser source code. It has more than 700 hundred packages [11].

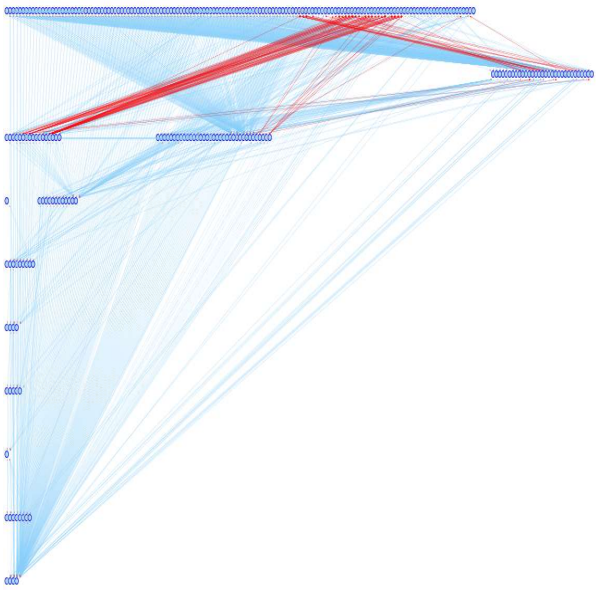

Figure 4. Sugiyama layout of Notepad++ package dependency.

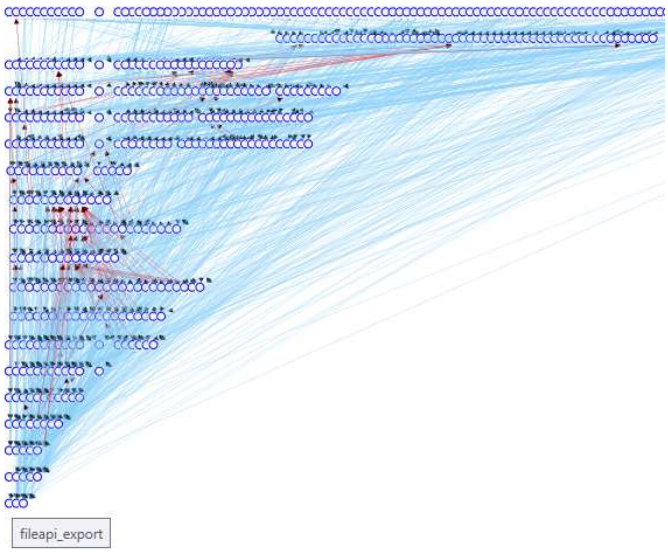


Figure 4. Sugiyama layout of Webkit package dependency.

### A. *Exploring Package Dependency Graphs*

Figure 4 and figure 5 show the dependency graph of Notepad++ and webkit respectively. One can explore dependency using features such as zooming, panning and viewing packages names from tooltips. In addition, one can explore areas of interest by selecting and zooming specific selections. In figure 3, the red arrows indicate edges going from lower to upper layers. Whereas blue arrows go from upper to lower layers. Generally, the red arrows are unwanted dependencies in a layered architecture as they result in cycle between layers. Similar observation can be made in Notepad++.

### B. *Layout After Applying Strong Components Clustering*

To collapse the red edges into one of the layers, strong components algorithm is run and the layout is redrawn using sugiyama layout. Figure 6 is the resulting dependency graph.

For sake of clarity some parts are clipped. The clustered nodes have more than one component. The tooltip text of one of the nodes is shown as an example. The red edges are significantly reduced. This is due to strong components algorithm clusters nodes in a cycle. Generally, we expect that the red edges to be contained in the clusters.

### C. *Clustering Each Layer Into A Node*

Further clustering nodes in each layer results in a graph that shows the relationship between the layers. This is achieved by adding an edge between layers if there is an edge going from any of the nodes from one layer to any other node in another layer. Figure 6 shows what we found for webkit.

Even though layering is generated automatically, after software designers examine its validity and manually restructure the dependency, they can generate rules, as discussed in the next section.

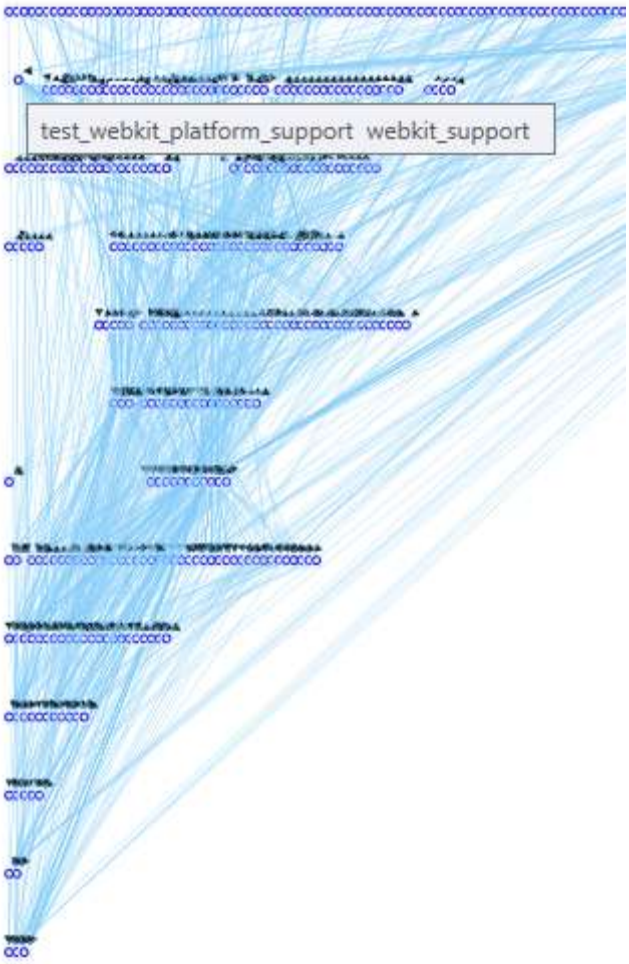


Figure 5. Webkit dependency visualization after applying clustering.

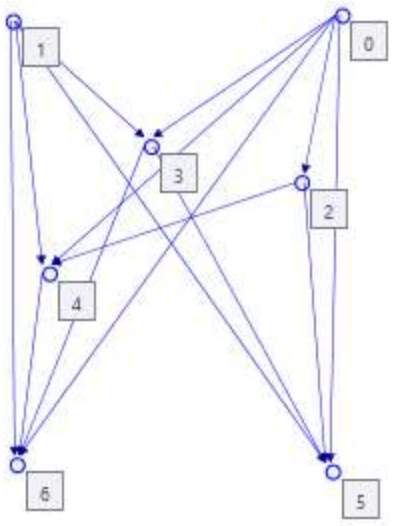


Figure 6. Layer dependency of webkit after clustering packages in each layer.

### D. Generating Dependency Rules

If engineers are satisfied with the above layering or the change they made manually after restructuring, they can automatically generate rules similar to figure 8.

Rules can have two parts. The ones shown in figure 8, for example are the ones that do not appear in figure 7. However, we can include implicit rules that prevent addition of dependency from lower layers to upper layers. Note that generating meaningful names for the nodes is beyond the scope of this work

Such rules can be used to notify engineers making change to the software system. This file can be changed by engineers whenever they want to restrict or relax the dependency structure.

```
<? xml version="1.0" encoding="UTF-8"?>
<graph title="Invalid Dependency">
 <node id="0">
    <Edge id="1"/> </node>
 <node id="1">
    <Edge id="2"/> </node>
 <node id="2"> <Edge id="3"/>
   <Edge id="6"/> </node>
 <node id="3">
   <Edge id="4"/> </node>
 <node id="4"> <Edge id="5"/> </node>
 <node id="5">  <Edge id="6"/> </node>
 <node id="6"> </node>
</graph>
```

Figure 7. Dependency rule.

## VI. Related Works

There are many works on software visualization in general. Source code-based visualization is done in [12], [13]. The work in [13] addresses some issues of understanding large industry size software. Class centered visualization is done in [14], [15], [16]. These works represent a class blue print to show the overall structure of a class, control flow among methods, and how methods access attributes. Software organization visualization has been done with optimized visual representation using trees [17], [18]. The organization can also be represented as a treemap [19], [20]. That means containment is defined in rectangular or circular spaces. Another aspect related to organization is concern for software components relationships. The most common ones are Dependency Structure Matrix (DSM), UML class diagrams and Simple Hierarchcal Multi-perspective(SHriMP) [21], [22]. From the three ways, ShHriMP is a relatively complete work. It shows software at source code level, class level and package level.

There are some research works on layering and cycles in software component dependency. Strong components in dependency is studied well in [2]. It emphasizes the fact that strong components make software maintainability and testability difficult. Some research is done to remove cycles to overcome this problem. A heuristic greedy algorithm to find the minimum feedback arc set is used [6]. We used this algorithm at the cycle removal stage of the Sugiyama layering algorithm.

Some research studies hierarchical organization of systems using graph algorithms. Sugiyama layout algorithm is used for hierarchical drawing of graphs [5].

A similar work to ours that visualizes object oriented programs is implemented as a polymeric view in [23]. It is different from our work in that it has a fine-grained view of objected-oriented programs. The approach represents metrics such as weighted calls per method and lines of code graphically. This work is extended in [24] by visualizing software programs at the package level.

In our work, layout algorithms such as spring layout and Sugiyama algorithms are used to show software component dependencies. Interactions such as zooming, panning and saving features allow the user to understand the diagram. Furthermore, Sugiyama layout algorithm is used to hierarchically represent dependencies. The layering and discrimination of edges is used to examine package dependency at a higher level. Strong component clustering along with Sugiyama layout algorithm is used to collapse unnecessary dependencies, red edges that may result in cycle, into a corresponding node in the nearby layer. It is also possible to cluster each layer after which rules to restrict dependency can be generated.

## VII. Conclusion and Future Work

We proposed a dependency visualization tool that can potentially be used to assess software design and generate dependency restriction rules. We implemented two layout algorithms that are used to explore package dependency of real software systems. Features including zooming, panning, pointed and selected zooming, saving layout and printing layouts are some of the functionalities of the tool. In addition, it helps one to explore high level design of a software and possibly help guide the restructuring of package level dependency.

Layering is one of the software architecture focused processes used in software design. The logical layering with additional input from engineers help restructure package dependency. After automatically identifying strong components in package dependency graphs, logically layering packages enables engineers to manage complexity by enabling them to control change in an ordered manner. Clustering packages in a layer after possible manual change of the dependency information results in a layered high level design of the software under investigation. If the layering in the high level layered design is found to be useful, rules can be generated that can be used as configuration file to check software change that prevents deterioration of design. Of course, such rules can be changed to further restrict or relax possible dependencies.

This work is a preliminary result. It has limitations that should be addressed in future work. Some of the planned works are:

- Getting feedback from real users will make the tool useful in real software design restructuring. The future change will be more concrete if we get feedback from engineers using our tool.
- Strong components hide cycles in package dependency. Whenever possible breaking these

cycles improves the design. Allowing automatic suggestions of such restructuring will be very helpful.

- Not all software designs benefit from layering architecture. Allowing other ways of arranging packages to automatically suggest high level design is also useful.

## REFERENCES


[1] Ultra –Large-Scale Systems: The Software Challenge of the Future, Retrieved from http://www.sei.cmu.edu/library/assets/uls_Book20062.pdf.

[2] J.W. Fawcett et al., Analyzing static structure of large software systems, proceedings of the 2005 International Conference on Software Engineering Research and Practice, 2005.

[3] T. Ball and S.G. Eick. Software visualization in the large, IEEE Computer, Vol. 29, April 1996, pp. 33–43.

[4] H. Gomma, Software Modeling and Design, Uml, Use cases, Patterns, and Software Architectures, Cambridge Univesity Press, 2011.

[5] K. Sugiyama and et al., Methods for Visual Understanding of Hierarchical System Structures, IEEE Transactions on Systems, Man, and Cybernetics, VOL. SMC- 1, NO. 2, 1981, pp. 109-125.

[6] P. Eades et al., A fast and effective heuristic for feedback arc set problem, Information processing letters, Vol 47, Issue 8, 1993, pp. 319 – 323.

[7] P.Eades. A heuristic for graph drawing. Congressus Nutnerantiunt, 42:149 – 160, 1984.

[8] Hu, Y. F. "Efficient, High-Quality Force-Directed Graph Drawing." The Mathematica Journal 10, no. 1 (2006): 37-71.

[9] T.M.J Fruchterman and E.M. Reigold, Graph drawing by force directed placement, Software – Practice and Experience, 21:1129 -1164, 1991.

[10] Notepad++, retrieved from http://notepad-plus-plus.org/download/v6.2.2.html , sept. 2012.

[11] Webkit from chromium web browser project, retrieved from http://dev.chromium.org/developers/how-tos/get-the-code , sept 2012.

[12] S. Eick, J. Steffen, and E. Summer Jr., "Seesoft – A Tool for Visualizing Line Oriented Software Statistics, "IEEE Trans. Software Eng., vol 18, no. 11, pp. 957-968, Nov. 1992.

[13] T. Ball and S. Eick, "Software Visulization in the Large," Computer, vol. 29, no. 4, pp. 33-43, Apr. 1996.

[14] M. Lanza, "Object Oriented Reverse Engineering – Coarse-Grained, Fine Grained, and Evolutionary Software Visualization, PhD dissertation, Univ. of Bern, 2003.

[15] M. Lanza and S. Ducasse, "A Catogorization of Classes Based on the Visualization of Their Internal Structure: The Class Blueprint," Proc. 16th ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications, pp. 300-311,200.

[16] S. Ducasse and M. Lanza, "The Class Blueprint: Visually Supporting the Understanding of Classes, "IEEE Tans. Software Eng. vol 31, no. 1, pp. 75 – 90, Jan. 2005.

[17] C. Wetherell and A. Shannon, "Tidy Drawings of Trees, "IEEE Trans. Software Eng., vol SE-5, no. 5, pp. 514 -520, Sept. 1979.

[18] T. Barlow and P. Neville, "A Comparison of 2D Visualization of Hierarchies, "Proc. IEEE Symp. Information Visualization, pp. 131-138, 2001.

[19] B. Johnson and B. Shneiderman, "Tree-Maps: A Space-Filling Approach to the Visualization of Hierarchical Information Structures, "Proc. Second IEEE Conf. Visualization, pp. 284 – 291, 1991.

[20] B. Shneiderman, "Tree Visualization with Tree-Maps: 2D Space-Filling Approach, "ACM Trans. Graphics, vol. 11, no. 1, pp. 92 – 99, Jan. 1992.

[21] M. Storey, H. Muller, and W.K., "Manipulating and Documenting Software Structures, "Software Visualization, pp. 244 – 263, World Scientific Publishing Co., 1996.

[22] M. Eiglsperger, "Automatic Layout of UML Class Diagrams: A Topology-Shape-Metrics Approach," PhD dissertation, Univ. Tubingen, 2003.

[23] R. Francese, M. Risi, G. Scanniello, and G. Tortora, "Proposing and assessing a software visualization approach based on polymetric views," *Journal of Visual Languages & Computing*, vol. 34–35, pp. 11–24, Jun. 2016.

[24] R. Francese, M. Risi, G. Scanniello, and G. Tortora, "Enhancing Polymetric Views with Coarse-Grained Views," in 2016 20th International Conference Information Visualisation (IV), 2016, pp. 57–62.

[25] J.W. Fawcett, (2016, December 10), Light Weight Parser [Online]. http://www.ecs.syr.edu/faculty/fawcett/handouts/WebPages/blogParser.htm.